Bioinformatics Advance Access published October 15, 2012

SortMeRNA: Fast and accurate filtering of ribosomal RNAs in metatranscriptomic data

Evguenia Kopylova *, Laurent Noé and Hélène Touzet

LIFL (UMR CNRS 8022, Université Lille 1) and Inria Lille Nord-Europe, 59655 Villeneuve d'Ascq, France

Associate Editor: Prof. Ivo Hofacker

ABSTRACT

Motivation: The application of Next-Generation Sequencing (NGS) technologies to RNAs directly extracted from a community of organisms yields a mixture of fragments characterizing both coding and non-coding types of RNAs. The tasks to distinguish among these and to further categorize the families of messenger RNAs and ribosomal RNAs is an important step for examining gene expression patterns of an interactive environment and the phylogenetic classification of the constituting species.

Results: We present SortMeRNA, a new software designed to rapidly filter ribosomal RNA fragments from metatranscriptomic data. It is capable of handling large sets of reads and sorting out all fragments matching to the rRNA database with high sensitivity and low running time.

Availability: `http://bioinfo.lifl.fr/RNA/sortmerna`

Contact: evguenia.kopylova@lifl.fr

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 INTRODUCTION

The application of NGS technologies for metatranscriptomic profiling has been a successful venture in practice. Scientists may now gain access to the full set of coding and non-coding RNA in a community of organisms, which becomes particularly important for samples which cannot be cultivated outside their native environment (Stewart *et al.*, 2011; Bomar *et al.*, 2011; Shi *et al.*, 2009). The initial challenge of metatranscriptomic sequenced data analysis is to sort apart the RNA fragments based on their biological significance. Messenger RNAs (mRNA) cast a universal glimpse on the gene expression patterns between interactive species. Likewise, the ribosomal RNAs (rRNA) disclose information on the community's structure, evolution and biodiversity, and prevail in classification and phylogenetic analyses. The rRNA can comprise up to 90% of total RNA. Various prior-to-sequencing procedures, such as mRNA amplification kits, can help to enrich the yield of mRNA (Gilbert and Hughes, 2011). However, these kits are not fully satisfactory since secondary steps may be required to verify if the resulting material is an accurate representative of the initial samples (Nygaard *et al.*, 2005). New software have been recently developed to address this issue, they identify and isolate rRNA fragments from a set of sequenced reads. The first set of programs, Meta-RNA 3 (Huang *et al.*, 2009), SSUALIGN (Nawrocki *et al.*, 2009) and rRNASelector (Lee *et al.*, 2011) share a common algorithmic approach, to represent an rRNA family database using a probabilistic model. Both Meta-RNA and rRNASelector use prebuilt Hidden Markov Models (HMM) and consequently sort reads against the database with the HMMER3 package (Eddy, 1998) whereas SSU-ALIGN uses Convariance Models to support secondary structure information. An alternative algorithm outside the domain of probabilistic models is riboPicker (Schmieder *et al.*, 2012) which uses a modified version of the Burrows-Wheeler Aligner (Li and Durbin, 2009). Lastly, BLASTN (Altschul *et al.*, 1990) is used in numerous home-made workflows for this problem. With BLASTN however, reads should be compared against all sequences of an rRNA database to achieve a good sensitivity level. In all cases, computational time is still an issue to handle large collections of reads.

In this paper we describe SortMeRNA, an efficient filter requiring only a representative set for an rRNA database and rapidly sorting through millions of reads. The underlying algorithm is analogous to the seeding strategy, focusing on finding many short regions of similarity between an rRNA database and a read. SortMeRNA also takes advantage of redundancy between homolog sequences, as HMMs do, and builds a compressed model of all rRNA sequences. The generated results adhere to the accuracy of the HMM-based programs and are computed in a fraction of the time.

2 SYSTEM AND METHODS

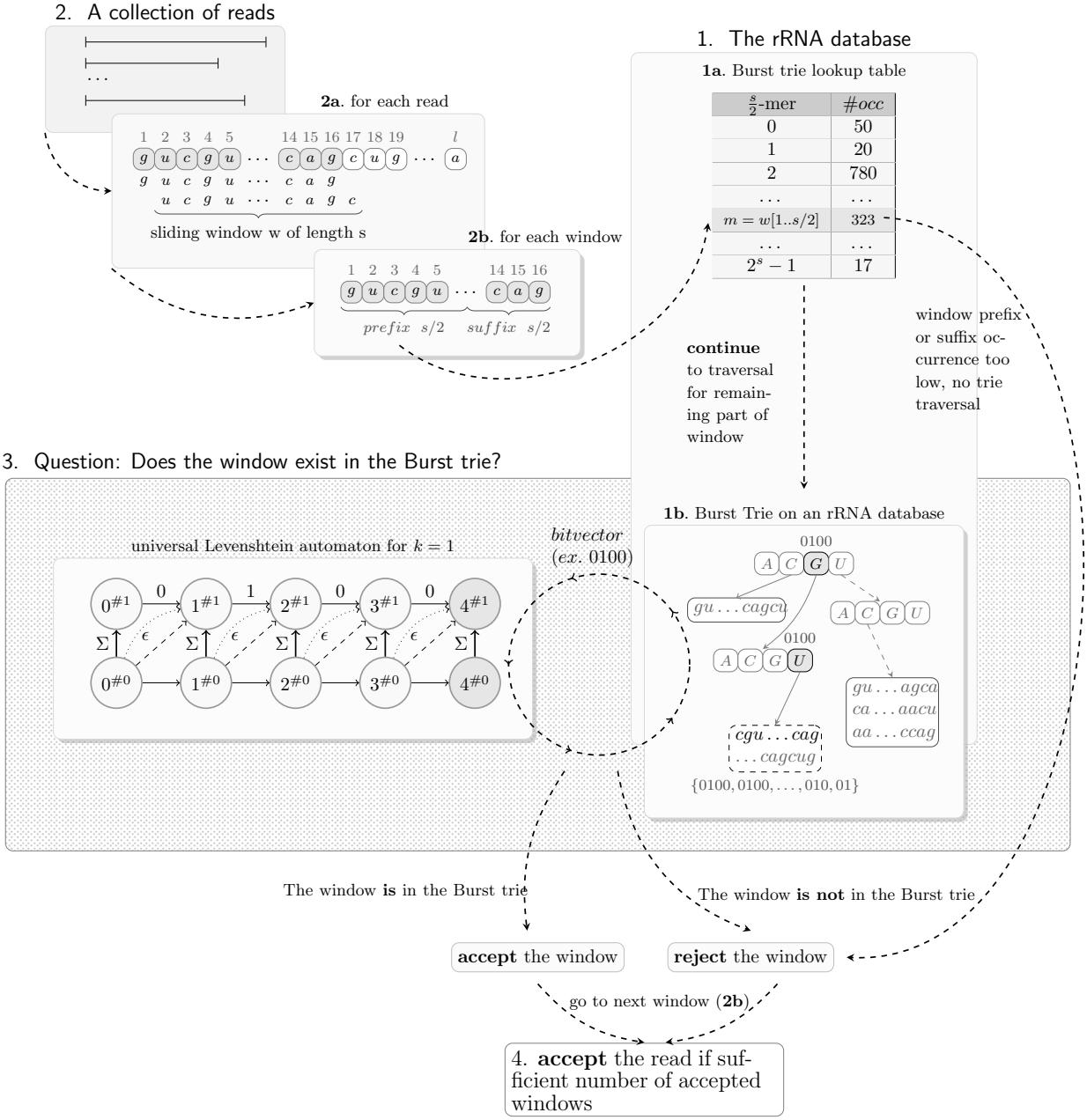
2.1 Algorithm overview

We assume having a collection of unassembled reads and a database of rRNA sequences, and we want to sort out reads that match to the database. The general principle behind our algorithm is to search for many short similarity regions between each read and the rRNA database. We scan each read with a sliding window, and the accepted reads are those which have more than a threshold number of windows present in the database. For a given read and a given window on the read, we authorize one error (substitution, insertion or deletion) between the window and the rRNA database.

To achieve this task in an efficient manner, the rRNA database is stored in a *Burst trie* coupled with a *lookup table* that speeds up the access to the Burst trie and takes advantage of conserved regions in the rRNA sequences. For a given read and a given window

*to whom correspondence should be addressed

Fig. 1. SortMeRNA Algorithm Overview. The set of representative rRNA sequences is preprocessed in the following way: **(1a)** The lookup table stores all of the $\frac{s}{2}$-mers and their number of occurrences which exist in the rRNA database. **(1b)** The Burst trie is a data structure which stores the rRNA database. **(2a)** The algorithm takes as input a collection of reads provided by the user, and for each read, a sliding window w of even length $s \in [14, 20]$ moves across the read. **(2b)** For each window w, the prefix $w[1..\frac{s}{2}]$ and suffix $w[s..\frac{s}{2}+1]$ are translated into a decimal value between 0 and $2^s - 1$. **(3)** If the value exists in the lookup table with a high frequency (see *Section 1.1* of the supplementary file), the remaining part of the window is searched in the Burst trie. This is done with a cyclic traversal between the universal Levenshtein automaton and the Burst trie that determines whether the subpattern is present in the rRNA database with at most 1 error. For every letter traversed in the Burst trie, a bitvector is passed to the universal Levenshtein automaton to verify if the number of encountered errors remains less or equal to 1. **(4)** After all windows have been traversed, if the number of accepted windows exceeds a certain threshold (see in *Section 2.5*) then the read is accepted and classified as rRNA.



on the read, we find the set of windows present in Burst trie using the *universal Levenshtein automaton*. This comparison is done by performing a *parallel traversal* between the Levenshtein automaton and the Burst trie.

Figure 1 globally illustrates this framework. The length s of the sliding window is a parameter of the algorithm, further discussed in *Section 2.5*. The acceptance of a read depends also on the ratio of matched windows. Let r be this parameter. This choice will also be discussed in *Section 2.5*.

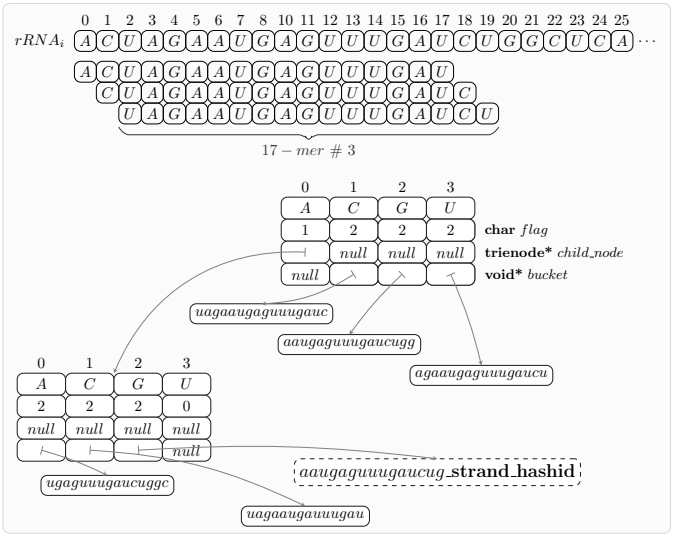
2.2 The Burst trie to store an rRNA database

The Burst trie (Heinz *et al.*, 2002) is a fast and versatile data structure which effectively stores a large number of strings such as an rRNA database. Unlike the standard trie, the binary search tree or other variants, which often adopt an equal rate of memory access among the cache or main memory, the Burst trie can exploit the modern cache architecture by addressing memory closest to the CPU. It is capable of reducing the number of trie nodes by 80% while maintaining performance similar to a hash table (Askitis and Zobel, 2010). Given a sequence vz, the Burst trie can store the prefix v as a link of trie nodes and the suffix z as an array of characters appended to the last trie node. Normally, subtrees become more sparse in the depth of a trie and representing them as reduced 'buckets' of contiguous memory preserves space and boosts cache-efficiency. When the number of sequences sharing a common prefix v reaches a fixed threshold, the appended bucket of suffixes bursts to form a new trie node and smaller sub-buckets. To optimize memory access during subtree traversal, the threshold size of a bucket should be less than the lower level cache. A systematic use of this trie can be observed in the fastest sorting algorithm for large sets of strings, the Burstsort (Sinha and Zobel, 2004).

Following a similar method of an array-structured trie as described in (Sinha *et al.*, 2006), our Burst trie is assembled exactly on the nucleotide alphabet $\{a, c, g, u\}$. As illustrated in Figure 2, the trie stores every unique $(s+1)$-mer substring in an rRNA database, since we look at windows of length s with at most 1 error between any two words. The information on whether the $(s+1)$-mer belongs to a forward strand, the reverse-complement or both (*strand*), and its origin (*hashid*) follows each entry in a bucket. When the exact location of the $(s+1)$-mer needs to be found in an rRNA database, the *hashid* value serves as an index in a complementary table storing this information. Nearly one-quarter of the 16S rRNA positions are $99-100\%$ conserved (Cannone *et al.*, 2002; Mears *et al.*, 2002) and this moderates the size of the trie since many identical or closely similar substrings are shared between sequences.

We use an additional optimization to improve access into the Burst trie. Since we consider at most one error between the window and the database, we have this simple property: For every two words such that the edit distance between them is bounded by 1, there exists a common substring of length $\frac{s}{2}$ which is either a prefix or a suffix of the two words. We apply this property to construct a lookup table storing all $\frac{s}{2}$-mers existing in the rRNA database. Note that for s in $[14, 20]$, transposing the nucleotide alphabet onto a binary equivalent, such that $\{a, c, g, u\} = \{00, 01, 10, 11\}$, we can represent each $\frac{s}{2}$-mer in s bits which maps to a unique integer value. Upon completion of the forward and reverse Burst tries, a scan of each trie is performed to record the existence of all $\frac{s}{2}$-mers and, if present, associated pointers to the trie node representing the

Fig. 2. Let $s = 16$, the Burst trie below is constructed on the first six 17-*mers* of an rRNA sequence. The '**char** $flag$' describes whether a pointer is set to a trie node '1', a bucket '2' or neither '0'. Additional information on the origin of the 17-mer directly follows each element, as shown in the dashed bucket.



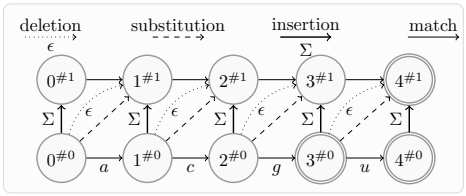
immediate letter following the prefix. The precomputed lookup table quickly determines whether an exact match of the prefix or suffix exists in the Burst tries and furthermore it provides us with direct access to the remaining part of the word in the Burst trie.

The lookup table also allows us to take into account distribution of $\frac{s}{2}$-mers in the rRNA database. A multiple sequence alignment of an rRNA database can clearly define areas of high nucleotide conservation and emphasize the evolutionary origins shared between organisms. In a similar manner, the lookup table defines highly conserved areas by keeping only frequent $\frac{s}{2}$-mer occurrences in the rRNA database. Before a window is traversed in the Burst trie, its prefix or suffix must exist in the lookup table. This notion enforces that a read matches closely to one region in a database rather than multiple scattered ones leading to a false alignment (see *Section 1.1* of the supplementary file).

2.3 The universal Levenshtein automaton

The classical nondeterministic Levenshtein automaton for a pattern p and a number of errors k recognizes the set of strings which are at most edit distance k to p (see Figure 3). This automaton is not suitable for computation because of the presence of multiple active states and epsilon transitions. This may be overcome by transforming the automaton into an equivalent deterministic form. However, the resulting automaton may be exponential in the length of p and likewise dependent on it. In (Schulz and Mihov, 2002) and (Mihov and Schulz, 2004), a universal Levenshtein automaton was characterized based upon insightful observations of the classical one. The term *universal* conveys its one-time construction and independency of p. The intuition arises from the symmetry of the nondeterministic automaton, which applies the same set of transition rules to every new input character and each new set of

Fig. 3. The nondeterministic Levenshtein automaton for $p = acgu$ and $k = 1$. The $s^{\#e}$ notation for each state corresponds to s number of characters read in the pattern p and e number of errors recorded. The initial state is $0^{\#0}$ and the three final states are $3^{\#0}$, $4^{\#0}$ and $4^{\#1}$. Each non-final state has three outgoing arcs, one for each type of edit operation.



active states is a subset of a known bounded superset. A set of bitvectors symbolizing the homology of p and a candidate string serve as input to the automaton. In full generality, the size of the automaton is exponential in a function of k (Mitankin, 2005). In our case, since $k = 1$, it remains sufficiently small. The set of bitvectors representing the similarity of two strings are precomputed using the following definition.

Definition 2.1. (Mihov and Schulz, 2004) The **characteristic vector** $\vec{\chi}(w, V)$ of a symbol $w \in \sum$ in a word $V = v_1 \dots v_n \in \sum^*$ is the bitvector of length n where the ith bit is set to 1 iff $w = v_i$.

The technical details of $n \leq 2k + 2$ and the prefix of k symbols of '$' appended to the pattern p can be found in the *(2004)* paper cited above.

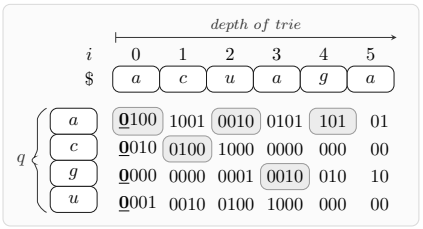
Example 2.1. Let $k = 1$, the input word $W = acaga$ and the pattern $p = \$acuaga$, then $\chi_1(\underline{a}, \$\underline{a}cu) = 0\underline{1}00$, $\chi_2(\underline{c}, a\underline{c}ua) = 0\underline{1}00$, $\chi_3(\underline{a}, cu\underline{a}g) = 00\underline{1}0$, $\chi_4(\underline{g}, ua\underline{g}a) = 00\underline{1}0$, $\chi_5(\underline{a}, \underline{a}g\underline{a}) = \underline{1}0\underline{1}$ are the computed characteristic bitvectors. It follows that $\{\chi_1, \dots, \chi_5\}$ is the *characteristic bitvector array* carrying the similarity information of x and p .

Beginning from χ_1 to $\chi_{|s|}$, the bitvectors are sequentially passed into the universal Levenshtein automaton. Each bitvector leads to a transition between states (in constant time) corresponding to the number of errors encountered thus far. If some χ_i reaches a failure state, greater than k errors exist between s and p, and the strings are rejected. The automaton only recognizes two strings if the input of the last bitvector $\chi_{|s|}$ leads to a final state.

2.4 Match of a read with the dynamic bitvector table

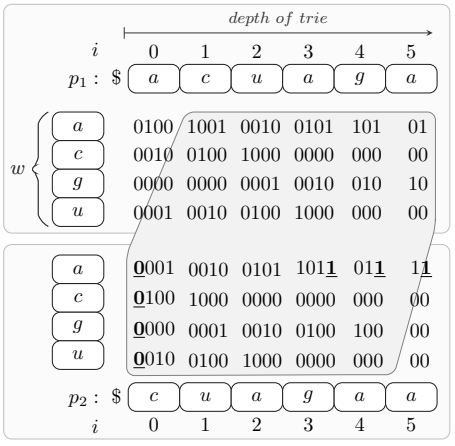
At this point, matching a window w of length s on the read against the rRNA database amounts to first checking whether the prefix or the suffix of length $\frac{s}{2}$ of w is present in the lookup table, then determining if the universal Levenshtein automaton for w recognizes some word in the Burst trie. For the second step, we have to implement a rapid traversal of the Levenshtein automaton which relies upon the precomputation of bitvectors for w. At every depth of the Burst trie, we assume that the symbol q in $\chi_i(q, V)$ appears as one of $\{a, c, g, u\}$ with equal probability. Ultimately during traversal, the bitvector of the actual residing nucleotide is chosen. Figure 4 shows the precomputation of bitvectors for $p = \$acuaga$ in **Example 2.1**. If the string $x = acaga$ existed in the trie, then the highlighted set of bitvectors $\{0100, 0100, 0010, 0010, 101\}$ would form the bitvector array (see *Section 1.2* of the supplementary file for a graphic example).

Fig. 4. The precomputed bitvector table for pattern $p = \$acuaga$ covering all possibilities of q for $k = 1$. The first bit in each entry of column $i = 0$ represents the $ symbol and is always set to '**0**'.



When the window is shifted by one position, the subsequent pattern p changes simply by the removal of the first character in the prefix and the addition of a new character in the suffix. Hence, rather than recomputing the bitvector table for each new window, a series of bitwise operations are taken to modify it, as demonstrated in Figure 5.

Fig. 5. The modification of the bitvector table from pattern $p_1 = \$acuaga$ to $p_2 = \$cuagaa$ for $k = 1$. Columns 0-2 of p_2 are equal to columns 1-3 of p_1, except for column 0, where the most significant bit (MSB) of every bitvector represents the symbol $ and is set to '**0**'. Column 3 of p_2 equals to column 4 of p_1 with an additional bit appended. The appended bit is set to '**1**' in the bitvector correspoding to the newly appended character, otherwise it is set to 0. Column 4 of p_2 is equal to column 3 of p_2, although the MSB is not considered. The same rule applies to column 5 of p_2, where the two MSBs of the column 3 bitvectors are not considered.



Following a pre-order path, the traversal of the Burst trie begins at the root node. Through knowledge of the nucleotide letter and the depth of the node being visited, the coinciding bitvector is accessed in the precomputed bitvector table, indifferent to whether the node is a trie node or a character in the bucket. Subsequently, the bitvector is passed to the universal Levenshtein automaton which decides

whether to continue traversal of the current subtree or backtrack to the first branching point with a non-failure Levenshtein state and recommence traversal of a new substree. In this manner, a complete traversal of the Burst trie remains unlikely as backtracking occurs each time the edit distance between the pattern and a traversed branch exceeds k. To further speed up Burst trie traversal for every window, a 'backwards dictionary' approach as described in (Mihov and Schulz, 2004) was implemented. The original algorithm builds two dictionaries, one for the forward strings and the second for their reverse equivalents. In this manner, the same window can be traversed quickly from both ends.

2.5 Parameter setting

The algorithm depends on two parameters: The length s of the sliding window, and the minimal proportion r of accepted windows in a read. To find a robust choice for s and r, we ran the algorithm for several values of s and r on several rRNA databases and for several sets of reads.

We purposely designed four databases with distinctive features: Small 16S and large 23S subunit, varying identity percentage and from distinct phylogeny tree subparts,

Set 1: 16S, 80% identity (2262 rRNA)
Set 2: 16S, 80% identity, truncated phylogeny tree (2187 rRNA)
Set 3: 23S, 95% identity (1969 rRNA)
Set 4: 23S, 95% identity, truncated phylogeny tree (1906 rRNA)

These databases were constructed by applying the ARB package (Ludwig *et al.*, 2004) and UCLUST (Edgar, 2010) to sequences from SILVA (Pruesse *et al.*, 2007) (see *Section 2.1* in the supplementary file). Next, we constructed datasets of simulated rRNA and non-rRNA reads using the software MetaSim (Richter *et al.*, 2008). We used two sequencing error models, Roche 454 and Illumina, because the errors for Roche 454 mainly originate as indels and for Illumina as substitutions. The length of the reads differs as well: $\geq$ 200nt for Roche 454 and 100nt for Illumina technology. To test the sensitivity on Set 1 and Set 3, we constructed 300,000 Roche 454 reads and 1,000,000 Illumina reads on the entire SILVA database minus the sequences used for the representative rRNA database. To measure the sensitivity for discovering new species with Set 2 and Set 4, the same number of reads was simulated only on the truncated sections of the Bacteria phylogeny tree. To test the selectivity, the non-rRNA reads were simulated using the NCBI bacterial genomes library with rRNAs masked (see *Section 2.2* in the supplementary file).

The parameter values were varied as: $s \in [14, 20]$ and $r \in (0, 1)$. The main conclusion is that $s = 18$, $r = 0.15$ for Roche 454 reads and $s = 18$, $r = 0.25$ for Illumina reads give best sensitivy/selectivity balance for all rRNA databases. Moreover, varying r within short ranges does not significantly affect the results (see *Section 2.3* of the supplementary file). We use these values as default settings in all subsequent analyses.

3 IMPLEMENTATION

SortMeRNA is implemented in C++ and freely distributed under the GNU general public license (GPL). It can be downloaded from `http://bioinfo.lifl.fr/RNA/sortmerna`. The software uses OpenMP functions to parallelize filtering of the reads. The input criteria are a fasta/fastq file of letter space reads produced by Roche 454 or Illumina technologies, and a fasta file of rRNA sequences. There are eight rRNA databases included in the software package covering the small (16S/18S), large (23S/28S) and 5/5.8S ribosomal subunit rRNAs, which were all derived from the SILVA and RFAM databases. Additionally, the user can work with their own RNA databases.

4 EXPERIMENTAL EVALUATION

The performance of SortMeRNA was measured in terms of sensitivity, selectivity and real-data analysis compared to the software SSU-ALIGN (Nawrocki *et al.* (2009)), Meta-RNA (Huang *et al.* (2009)), rRNASelector (Lee *et al.* (2011)), riboPicker (Schmieder *et al.* (2012)) and BLASTN (Altschul *et al.* (1990)). All tests were performed on an Intel(R) Xeon(R) CPU W3520 2.67GHz machine with 8GB RAM, L1 cache size of 32 KB, L2 cache size of 256 KB and L3 cache size of 8192 KB. Since riboPicker and SSU-ALIGN do not provide a direct option for multi-threading, all tests were carried out using one thread.

4.1 rRNA databases

We created two new representative databases: 16S rRNA with 85% identity (7,659 sequences) and 23S rRNA with 98% identity (2,811 sequences) (see *Section 3.2* of the supplementary file). The 16S rRNA database was used by SortMeRNA, riboPicker, BLASTN and SSU-ALIGN, and the 23S rRNA database by SortMeRNA, riboPicker and BLASTN. SSU-ALIGN was written for aligning small ribosomal subunits and does not provide models for 23S rRNA. riboPicker was also tested with a more comprehensive database made available from their website: All 16S and 23S rRNA sequences taken from SILVA, RDP-II, Greengenes, NCBI archaeal and bacterial genomes, and HMP (3,232,371 16S and 19,602 23S unique sequences). The results for this larger database are indicated by riboPicker* in the subsequent tables. For Meta-RNA and rRNASelector, we used the HMMs provided with the software.

4.2 Simulated reads

Sensitivity for 16S rRNA. 300,000 Roche 454 and 1,000,000 Illumina 16S rRNA reads were simulated in the same manner as described in *Section 2.5*. The performance results can be viewed in Table 1. All software programs except riboPicker and SSU-ALIGN have a sensitivity level higher than 97%, and even higher than 99% for BLASTN and SortMeRNA. The sensitivity for riboPicker is very low (56%) because BWA-SW works well with error rates 2%-3% for 100-200nt reads, and loses sensitivity for new species. As expected, the sensitivity increases with a larger database (indicated riboPicker*). Considering the computation time, SortMeRNA runs in less than 2 minutes, or 72x faster than the next fastest tool with proportionate sensitivity (Meta-RNA). Note also that BLASTN executes at a very slow speed (several hours), because reads should be compared against all of sequences in the representative database.

Selectivity for 16S rRNA. 1,000,000 Roche 454 and 1,000,000 Illumina non-16S rRNA reads were simulated in the same manner as described in *Section 2.5*. The performance results can be viewed

in Table 2. All programs have a selectivity level higher than 99.98%. The number of false positives for the HMM-based programs remains comparable to SortMeRNA for both Illumina and Roche 454 reads. The difference in the simulated data results between Meta-RNA and rRNASelector can be attributed to the number of bacteria vs. archaea rRNA sequences used in the construction of the HMMs, as well as additional parameter settings in rRNASelector. riboPicker* and BLASTN show the lowest selectivity. Concerning the running time, the order of the fastest programs is rRNASelector, Meta-RNA and SortMeRNA. Both rRNASelector and Meta-RNA use the HMMER3 package, which applies a pre-filter to quickly reject sequences which would score very low in the HMM. This acceleration heuristic gives these programs a competitive advantage on the artificial dataset for selectivity where all of the sequences are negative.

Results for 23S rRNAs are analogous in terms of accuracy and running time. They can be found in Table A and Table B under *Section 3.3* of the supplementary file.

4.3 Real data

The metatranscriptomic datasets SRR106861 of a photosynthetic microbial community and SRR013513 of a tidal salt marsh creek from 454 sequencing were downloaded from the NCBI Sequence Read Archive. The results for 16S and 23S can be viewed in Table 3 and Table 4 respectively, and the overlap of the results between tools in Figure 6 and Figure 7. The results obtained with SortMeRNA are very close to the ones obtained with HMM-based methods. riboPicker finds only a fraction of all potential rRNAs, which confirms its low sensitivity for small databases. The majority of 16S reads found only by riboPicker* (1,298) map to mRNA. For 23S analysis in Table 4 and Figure 7, approximately 99% of the excess reads of Meta-RNA (12,112) and rRNASelector likewise map to 28S, along with 83% of the (624) reads found only by BLASTN and Meta-RNA. The (537) reads found only by BLASTN map to 16S rRNA, ncRNA and mRNA.

5 DISCUSSION

SortMeRNA has shown to be a rapid and efficient filter which can sort a large set of metatranscriptomic reads with high accuracy comparable to the HMM-based programs. SortMeRNA implements seeds with errors (substitution and indel) and this important characteristic renders the algorithm robust to errors of different types of sequencers while providing the ability to discover new rRNA sequences from unknown species.

The method used by the algorithm is universal and flexible. The database can be constructed on any family of sequences provided by the user. Moreover, the algorithm does not require a multiple sequence alignment file to build the database, as HMM-based programs do, and this is an advantage when sequences are hard to align or only partial sequences are available. Another advantage of SortMeRNA is the small number of parameter settings required by the program (see *Section 2* of the supplementary data).

Table 3. Runtime for the SRR106861 metatranscriptome of 105,873 reads against a 16S rRNA database of 7,659 sequences.

	rRNA	run time	latency	memory (%)
SortMeRNA	27046	34s	1x	4.8
riboPicker	11389	4m10s	7x	2.3
riboPicker *	27195	39m3s	69x	30.8
BLASTN	27061	1h29m	157x	0.6
Meta-RNA	27111	10m33s	18x	1.8
rRNASelector	27085	10m40s	18x	0.8

Fig. 6. Venn diagram for reads classified as 16S rRNA by BLASTN, Meta-RNA, SortMeRNA and riboPicker* in the SRR106861 metatranscriptome.

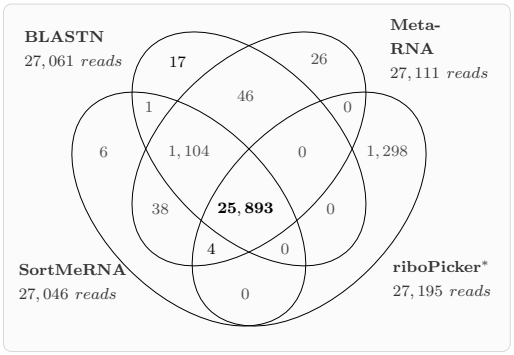


Table 4. Runtime for the SRR013513 metatranscriptome of 207,368 reads against a 23S rRNA database of 2,811 sequences.

	rRNA	run time	latency	memory (%)
SortMeRNA	94395	51s	1x	3.8
riboPicker	71937	10m2s	12x	3.9
riboPicker *	84152	36m27s	43x	5.5
BLASTN	94439	3h42m	261x	0.9
Meta-RNA	106698	1h33m	109x	4.8
rRNASelector	107900	1h36m	113x	3

Fig. 7. Venn diagram for reads classified as 23S rRNA by BLASTN, Meta-RNA, SortMeRNA and riboPicker* in the SRR013513 metatranscriptome.

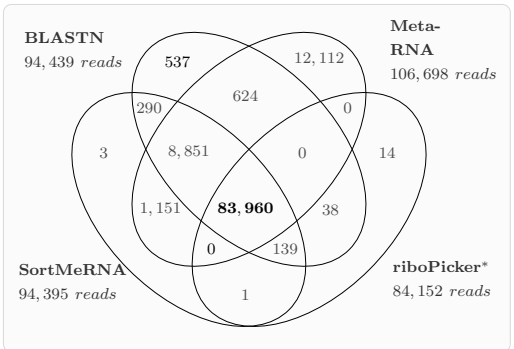


Table 1. SENSITIVITY. 1,000,000 of MetaSim simulated Illumina ($100nt$) and 300,000 Roche 454 ($\geq 200nt$) **rRNA** reads against a representative 16S rRNA database of 7,659 sequences.

	Illumina					Roche 454				
	rRNA	run time	latency	memory (%)	sensitivity (%)	rRNA	run time	latency	memory (%)	sensitivity (%)
$SortMeRNA$	998615	1m39s	1x	8.5	99.861	299979	1m43s	1x	6.3	99.993
$riboPicker$	558607	18m45s	11x	6.8	55.860	123024	18m36s	11x	5.6	41.008
$riboPicker$ *	999941	6h33m	238x	35.3	99.994	299999	9h	314x	34	99.999
$BLASTN$	995322	23h52m	868x	3.0	99.532	299978	18h35m	649x	1.4	99.992
$Meta\text{-}RNA$	983332	2h	72x	33.3	98.333	299980	1h57m	68x	12.9	99.993
$rRNASelector$	974118	1h47m	64x	17.4	97.411	299976	2h	70x	7	99.992
$SSU\text{-}ALIGN$	971221	6h49m	248x	0.1	97.122	299902	5h50m	204x	0.1	99.967

Table 2. SELECTIVITY. 1,000,000 of MetaSim simulated Illumina ($100nt$) and 1,000,000 Roche 454 ($\geq 200nt$) **non-rRNA** reads against a representative 16S rRNA database of 7,659 sequences.

	Illumina					Roche 454				
	rRNA	run time	latency	memory (%)	selectivity (%)	rRNA	run time	latency	memory (%)	selectivity (%)
$SortMeRNA$	17	2m9s	2x	7.6	99.9983	13	3m42s	1x	10.2	99.9987
$riboPicker$	7	10m22s	8x	6.7	99.9993	3	29m45s	9x	16.8	99.9997
$riboPicker$ *	158	56m37s	42x	35.1	99.9842	53	2h43m	49x	45.2	99.9947
$BLASTN$	33	14m22s	11x	0.3	99.9967	33	16m12s	5x	0.3	99.9967
$Meta\text{-}RNA$	11	1m33s	1x	0.1	99.9989	11	3m41s	1x	0.2	99.9989
$rRNASelector$	10	1m20s	1x	0.1	99.9990	11	3m21s	1x	0.2	99.9989
$SSU\text{-}ALIGN$	8	3h51m	173x	0.1	99.9992	11	10h30m	188x	0.1	99.9989

ACKNOWLEDGEMENTS

This research was supported by ANR project MAPPI (ANR-2010-COSI-004), LIFL (UMR CNRS 8022 Université Lille 1) and Inria Lille Nord-Europe. Project MAPPI is associated with the Tara Oceans expedition (`oceans.taraexpeditions.org`), where the principal tasks involve the development of new software for mapping and assembling metagenomic and metatranscriptomic data.

REFERENCES

Altschul, S. *et al.* (1990). Basic local alignment search tool. *J. Mol. Biol.*, **215**, 403–10.

Askitis, N. and Zobel, J. (2010). Redesigning the string hash table, burst trie, and bst to exploit cache. *ACM JEA*, **15**.

Bomar, L., Maltz, M., Colston, S., and Graf, J. (2011). Directed culturing of microorganisms using metatranscriptomics. *mBio*, **2**(2).

Cannone, J. *et al.* (2002). The comparative rna web (crw) site: An online database of comparative sequence and structure information for ribosomal, intron, and other rnas. *BMC Bioinformatics*, **3**, 15.

Eddy, S. (1998). Profile hidden markov models. *Bioinformatics*, **14**, 755–763.

Edgar, R. (2010). Search and clustering orders of magnitude faster than blast. *Bioinformatics*, **26**, 2460–2461.

Gilbert, J. and Hughes, M. (2011). Gene expression profiling: metatranscriptomics. *Meth. Mol. Biol.*, **733**, 195–205.

Heinz, S., Zobel, J., and Williams, H. (2002). Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, **20**, 192–223.

Huang, Y., Gilna, P., and Li, W. (2009). Identification of ribosomal rna genes in metagenomic fragments. *Bioinformatics*, **25**, 1338–1340.

Lee, J., Yi, H., and Chun, J. (2011). rrnaselector: a computer program for selecting ribosomal rna encoding sequences from metagenomic and metatranscriptomic shotgun libraries. *J. Microbiol.*, **49**, 689–91.

Li, H. and Durbin, R. (2009). Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, **25**, 1754–60.

Ludwig, W. *et al.* (2004). Arb: A software environment for sequence data. *Nucl Acids Res*, **32**, 1363–1371.

Mears, J. *et al.* (2002). Modeling a minimal ribosome based on comparative sequence analysis. *J. Mol. Biol.*, **321**, 215–34.

Mihov, S. and Schulz, K. (2004). Fast approximate search in large dictionaries. *J. Comput. Ling.*, **30**, 451–477.

Mitankin, P. (2005). *Universal Levenshtein Automata. Building and Properties*. Master's thesis, Sofia University, Bulgaria.

Nawrocki, E., Kolbe, D., and Eddy, S. (2009). Infernal 1.0: inference of rna alignments. *Bioinformatics*, **25**, 1335–7.

Nygaard, V. *et al.* (2005). Limitations of mrna amplification from small-size cell samples. *BMC Genomics*, **6**(147).

Pruesse, E. *et al.* (2007). Silva: a comprehensive online resource for quality checked and aligned ribosomal rna sequence data compatible with arb. *Nucl. Acids Res.*, **35**, 7188–7196.

Richter, D., Ott, F., Auch, A., Schmid, R., and Huson, D. (2008). A sequencing simulator for genomics and metagenomics. *PLoS ONE*, **3**.

Schmieder, R., Lim, Y., and Edwards, R. (2012). Identification and removal of ribosomal rna sequences from metatranscriptomes. *Bioinformatics*, **28**, 433–435.

Schulz, K. and Mihov, S. (2002). Fast string correction with levenshtein automata. *IJDAR*, **5**, 67–85.

Shi, Y., Tyson, G., and DeLong, E. (2009). Metatranscriptomics reveals unique microbial small rnas in the ocean's water column. *Nature*, **459**(7244).

Sinha, R. and Zobel, J. (2004). Cache-conscious sorting of large sets of strings with dynamic tries. *ACM JEA*, **9**.

Sinha, R., Zobel, J., and Ring, D. (2006). Cache-efficient string sorting using copying. *ACM JEA*, **11**.

Stewart, F. *et al.* (2011). Metatranscriptomics analysis of sulfur oxidation genes in the endosymbiont of solemnya velum. *Front. Microbiol.*, **2**(134).